Design, Implementation and Verification of Cloud Architecture for Monitoring a Virtual Machine's Security Health

Tianwei Zhang, Ruby B. Lee

Abstract—Cloud customers need guarantees regarding the security of their virtual machines (VMs), operating within an Infrastructure as a Service (IaaS) cloud system. This is complicated by the customer not knowing where his VM is executing, and on the semantic gap between what the customer wants to know versus what can be measured in the cloud. We present *CloudMonatt*, an architecture for monitoring a VM's security health. We show a full prototype based on the OpenStack open source cloud software.
It is necessary to verify *CloudMonatt* to guarantee that there are no security vulnerabilities that could allow an attacker to subvert its protection. As such, we conduct a systematic security verification of *CloudMonatt*. We model and verify the network protocols within the distributed system, as well as interactions of hardware/software modules inside the cloud server. Our results show that *CloudMonatt* is capable of delivering this monitoring and attestation service to customers in an unforgeable and reliable manner.

Index Terms—Cloud Computing, virtual machine, security health, attestation, security verification

1 Introduction

In an IaaS cloud, a customer requests to launch a VM in the cloud system. During the VM's lifetime, the customer would like to know if his VM has good security health.

Security Health of a VM is defined as an indication of the likelihood that the VM satisfies the security properties the customer requested for his leased VM. It depends on a variety of factors in the complicated cloud environment. First, a VM can get infected with malware or OS rootkits at runtime. Such *inside-VM* vulnerabilities can take complete control of the VM and significantly compromise its security state. Second, cloud management software usually have large code base sizes. This inevitably introduces bugs and gives adversaries opportunities to conduct privilege escalation attacks and gain root privilege [1]. Then the adversaries have full control of the whole server, as well as the capability of compromising any VM's security health on this server. Third, cloud systems usually adopt the "multi-tenancy" feature, where different customers share the same cloud server, as co-tenants or co-resident VMs. Past work have shown that the "bad neighbor" VMs are able to steal critical information through side-channel attacks [2], [3], thus compromising the VM's *confidentiality health*, or steal computing resources through Resource-Freeing attacks [4] or Memory DoS attacks [5], thus compromising the victim VM's *availability health*. We call the threats from the host OS and co-located VMs *outside-VM* vulnerabilities, which are hard for customers to defeat. Hence, a VM's security health depends on not only the activities inside the VM, but also the VM's interactions with its environment.

Monitoring the VMs' security health poses a series of challenges in a cloud system. First, the customer's limited privileges prevent him from collecting comprehensive security measurements to monitor his VM's health securely. He only has access to the VM, but not to the host server. For *inside-VM* vulnerabilities, once the VM's OS is compromised by the attacker, the customer may not get correct measurements. For *outside-VM* vulnerabilities, the customer cannot collect information about the co-resident VMs, hypervisor, etc. Second, the customer's desired security requirements are expressed in terms of a VM, but the security measurements usually involve the physical server, the hypervisor and other entities related to this VM. This creates a semantic gap between what the customers want to monitor and the type of measurements that can be collected. Third, the VMs go through different lifecycle stages and may migrate to different host servers. A seamless monitoring mechanism throughout the VMs' lifetime is therefore highly desirable. Fourth, there are numerous entities between the customers and the point of VM operations. It is important to collect, filter and process the attestation information securely to attest, i.e., pass on to the customer in an unforgeable way, only the requested information.

To address these challenges, Zhang and Lee proposed a flexible architecture, *CloudMonatt*, to monitor and attest the security health of customers' VMs within a cloud system [6]. *CloudMonatt* is built upon the *property-based* attestation model, and provides several novel features. First, it provides a framework for monitoring different aspects of security health. Second, it shows how to interpret and map actual measurements collected to security properties that can be understood by the customer. These bridge the semantic gap between requested VM properties and the platform measurements for security health. Third, attestations can be done at runtime and for VM migrations, not just at boot up and VM launch time. Fourth, *CloudMonatt* provides remediation responses based on the monitored results.

This paper is an extension over the *CloudMonatt* work [6]. First we describe the design (Section 3) and implementation

• *T. Zhang and R. B. Lee are with the Department of Electrical Engineering, Princeton University, Princeton, NJ, 08544.*
E-mail: {tianweiz, rblee}@princeton.edu

(Section 4) of *CloudMonatt* [6]. Then we demonstrate the security verification of this architecture. Given that *CloudMonatt* is designed to monitor and report VMs' security health, it is important and necessary to systematically check that it works correctly as expected, with no vulnerabilities that could be exploited by attackers to subvert its security scheme. A distributed cloud system like *CloudMonatt* involves a variety of cloud servers, and hardware, software and network components. This requires us to consider the external communication protocols between servers, as well as the internal activities inside each server.

We adopt the methodology from [7] to demonstrate how to practically verify a distributed cloud system. Specifically, we break down the whole verification task into two parts, the external network verification and the internal server verification. We model components as state machines, propose the security invariants for checking, and use a protocol checking tool, *ProVerif* [8] to model the network protocols and system operations, and verify the invariants. The verification results not only raise our confidence in our *CloudMonatt* design, but also provide suggestions for further strengthening its reliability and trustworthiness.

Key contributions in this paper are:

- Design and implementation in OpenStack, of a flexible architecture to monitor the security health of VMs over the VMs' lifecycle, with automatic remediation response to failing security health indicated by negative attestation results.
- A systematic security verification of this secure cloud architecture. We break the verification task of the distributed architecture into external and internal verification, propose security invariants for each task, and repurposing existing tools to verify these invariants.

Section 2 reviews the background and related work. Section 3 describes the *CloudMonatt* architecture and its operations. Section 4 gives the details of our prototype implementation. Section 5 shows our proposed security evaluation. We conclude in Section 6.

2 Background and Related Work

2.1 Remote Attestation

Remote attestation is defined as *"the procedure of making claims about the security conditions of a targeted system based on the evidence supplied by that system"* [9]. It often involves three entities: an *attester* is the targeted system which provides the evidence; a *verifier* is an entity which requests a report for a given attester; an *appraiser* is an entity which makes decisions by evaluating the security conditions based on the attester's evidence. We review two types of attestations.

Binary attestation. Proposed by the Trusted Computing Group (TCG) [10], binary attestation was a breakthrough development which helped enable attestation of platform integrity of a remote server. The attester calculates binary hash values of the platform configurations, and sends them to the verifier. The verifier, who typically also plays the role of the appraiser, compares these values with reference or "good" configurations, and determines whether the state of the attester is acceptable.

Many systems enabled with remote attestations have been designed, based on the Trusted Computing Group's binary attestation [10]. Sailer et al. [11] proposed the Integrity Measurement Architecture (IMA), to measure the integrity of executables from BIOS to application level. Jaeger et al. [12] extended IMA to Policy Reduced Integrity Measurement Architecture (PRIMA), which can measure the Mandatory Access Control (MAC) Policy defined for controlling information flows across user processes. Shi et al. [13] proposed the architecture of Binding Instructions aNd Data (BIND) to realize fine-grained attestation on the integrity of code in distributed systems. Seshadri et al. [14] proposed the Pioneer system, which can attest the integrity of code executions on legacy computing systems. Garfinkel et al. [15] designed Terra, an architecture with a trusted virtual machine monitor (TVMM) to provide a secure computing environment by isolating critical application code in different VMs.

Binary attestation has certain shortcomings [16], [17]. First, binary measurements sent to the verifier provide configuration and implementation details of the attester, which is a privacy issue and may lead to fingerprinting attacks. Second, the verifier (who is also the appraiser) must be aware of the correct configurations of the target platform. Third, the target platforms may get updated leading to a change in configurations, and thus requiring the verifier to be notified about it each time.

Property-based attestation. To address the above shortcomings, property-based attestation was proposed [16], which attests security properties, functions and behaviors of systems. In property-based attestation, the verifier and the appraiser are separate entities. The appraiser is a trusted third party, who is trusted by the attester and the verifier. The appraiser has full knowledge of the attester. Its job is to transform the attester's measurements into properties and vice versa, and determine if the attester satisfies a set of given properties. A common solution for realizing an appraiser's interpretation mechanism is delegation-based attestation [16], [18]. In this approach, the appraiser can issue a *property certificate*, proving that a given configuration fulfills a specific property demanded by the verifier. Other approaches like proxy-based [19] are also proposed and implemented.

A variety of methods deriving from *property-based attestation* are explored, to attest different security properties. Haldar et al. [20] proposed semantic attestation, which monitors programs' high-level dynamic behaviors and properties. Alam et al. [21] proposed model-based behavioral attestation to attest the behaviors of security policies associated with the platforms. Sirer et al. [22] proposed logical attestation, which translates the programs' high-level attributable properties to logical expressions for verification.

Compared with binary attestation, the advantages of property-based attestation are as follows: properties do not reveal the configuration and implementation details, and thus do not violate the privacy of the attester; properties do not change as often as the target platform's configurations; properties are easier to understand and express. However, the specification and interpretation of properties to be attested remain as challenging, open problems [17]. They make it very difficult for computer architects to convert the concept of *property-based attestation* into real architectures.

2.2 Attestation in Cloud Computing

Attestation of VM security health in the cloud environment is more complex. In IaaS, the *verifier* is the customer who launches a VM in the cloud and the *attester* is the VM. The health of the target VM depends on not only the applications and OS within the VM, but also its interactions with the host environment. In addition, since a VM experiences different activities during its lifecycle, it is important to consider attestation throughout the VM's life.

Virtual machine introspection. Past work on *inside-VM* threats proposed virtual machine introspection techniques [23], [24], [25], [26], [27], [28]. The hypervisor monitors the VM and detects the existence of malicious entities inside the VM, while being isolated, and thus protected, from the VM.

However, these methods detect abnormal behaviors inside the VM, but do not consider the threats from co-resident VMs or other outside-VM entities. Also, how to use these techniques in the cloud system and allow the remote customer to use these monitoring services are problems which have not been addressed. In [6], we show how virtual machine introspection can be seamlessly deployed in our *CloudMonatt* architecture to monitor a VM's integrity.

Direct attestation. Direct attestation allows the customers to talk to the VMs directly. One example is the virtual Trusted Platform Module (vTPM) [29], [30], [31], [32], [33]. Since a physical TPM cannot be directly used by the VMs within virtualized environments, vTPMs are designed to provide the same usage model and services to the VMs. Then attestation can be carried out directly between the customers and virtual machines by the vTPM instances. These instances can be realized by implementing TPM emulators in the hypervisor or host OS, by modifying the hardware TPM to enable TPM virtualization [29], or by combining both software and hardware TPMs [30]. To overcome binary attestation's shortcomings, Sadeghi et al. [31] proposed virtual property-based attestation, in which the vTPM instances are assigned the tasks of security property management and interpretations.

The virtual TPM solution raises some problems for VM monitoring: it cannot attest the security conditions of the VM's environments. Furthermore, the monitoring tool resides in the guest OS, so it needs modification of the guest OS, and commodity OSes are also highly susceptible to attacks.

Centralized attestation. To overcome the above problems, the concept of *centralized attestation* is introduced in the cloud system to manage the attestation procedure. Schiffman et al. [34] implemented a centralized "cloud verifier" that can provide the integrity attestations for customers' VM applications. Customers issue the authorization for the VM to access applications only when the integrity attestation passes. Santos et al. [35] designed a centralized monitor to check the platform's configurations and map them to security attributes. This enables customers' VMs to be allocated on the platforms with specified attributes. Then Attribute-Based Encryption is exploited to seal and unseal data between customers and cloud servers to ensure they are not compromised. However, the above work are still based on *binary attestation* for platform integrity and configuration checking, and do not consider other security properties like confidentiality or availability, nor the VMs' interactions (intended or unintended) with the outside-VM environment.

In contrast, this paper uses *centralized property-based attestation* to monitor both inside-VM and outside-VM health. We introduce a centralized Attestation Server which conducts security monitoring management, property translation and interpretation. On each cloud server we introduce a `Monitor Module` integrated with many monitoring tools to monitor VMs for different properties. These new designs can achieve a comprehensive *property-based* attestation and monitoring of VMs' health in clouds. We enable attestation not only on boot up and VM initiation, but also during VM runtime and migration. We also propose a novel ongoing periodic attestation for a VM's security health, and automated remediation responses for negative attestation results. Concrete examples of monitoring VMs' different security properties can be found in our past work [6].

2.3 Attestation Protocols

Remote attestation needs the support of cryptographic protocols. For *binary attestation*, the most basic protocol is the standard signature scheme which was originally adopted by TCG (TPM specification v1.2) [10], [11]. Manufacturers burn a private key into the micro-processor chip, and then the TPM generates the attestation key-pairs, signs the public key, and sends it to the privacy certificate authority for a certificate. The TPM specification v1.2 also includes the Direct Anonymous Attestation functionality [36], which can preserve the anonymity of the attested platforms from the verifier using the group signature scheme. Then TCG released TPM specification v2.0 [37], [38], which included multiple cryptographic functionalities and flexibly selective cryptographic algorithms, e.g., anonymous signatures, pseudonym signatures, and conventional signatures. Stumpf et al. [39] enhanced the TCG-based attestation protocol by integrating Diffie-Hellman key exchange protocol to defeat masquerading attacks.

For *property-based attestation*, Chen et al. [18] designed a provable and efficient protocol with a delegation solution. This protocol holds the security features of unforgeability (i.e., the signature can only be produced by the valid TPM) and unlinkability (i.e., the verifier cannot deduce the specific configuration of the platform). It also supports the revocation of invalid certifications. Chen et al. [40] proposed another property-based attestation protocol based on the ring signature scheme. This protocol can preserve the platform's privacy and avoid the involvement of a trusted third party to certify properties, which will be done by the attested platform. Different from past work, the protocol in *CloudMonatt* involves four entities in a cloud system. We design new protocols to provide unforgeability for attestation reports, and anonymity for attested platforms. Anonymity will be discussed in Section 3.4.2.

3 CloudMonatt Architecture

In this section we describe *CloudMonatt*, a flexible distributed cloud architecture that can monitor the security health of the customers' VM in the cloud, detect the vulnerabilities inside the VM, from the platform it is running on or from co-resident VMs, and take prompt remediation actions when the VM's security health is appraised as inadequate. Section 3.1 describes the main architectural components. Section 3.2 describes the threat model, referring to these components. Section 3.3 demonstrates the definition of VM security health and security property interpretation. Section 3.4 describes the protocols to guarantee the attestation unforgeability.

3.1 Architecture Overview

Figure 1a shows an overview of the *CloudMonatt* architecture. This includes four entities: 1) Cloud Customer, 2) Cloud Controller, 3) Attestation Server and 4) Cloud Server.

3.1.1 Cloud Customer

The customer is the initiator and end-verifier in the system. He places a request for leasing VMs with specific resource requirements and security requests to the Cloud Controller. He can issue any number of security attestation requests during his VM's lifetime. *CloudMonatt* allows customers to invoke the monitoring and attestation requests at any time during the VM's lifecycle.

3.1.2 Cloud Controller

The Cloud Controller acts as the cloud manager, responsible for taking VM requests and servicing them for each customer. The `Policy Validation Module` in the Controller selects qualified servers for customers' requested VMs. These servers need to both satisfy the VMs' demanded physical resources, as well as support the requested security properties and their property monitoring services. The `Deployment Module` allocates each VM on the selected server.

During the VMs' lifecycle, the customers may request the Cloud Controller to monitor the security properties associated with their VMs. The Cloud Controller will entrust the Attestation Server to collect the monitored security measurements from the correct VMs, and send a report back to it. It then sends the results back to the customers to keep them informed of their VMs' security health.

When these results reveal potential vulnerabilities for the VMs, the `Response Module` in the Controller carries out appropriate remediation responses. When an inside-VM vulnerability is detected, the Controller can shut down the VM to protect it from attacks. If the security health of the current server is questionable, the Controller can temporarily suspend the VM and then resume it when the server has returned to the desired security health. The Controller can also migrate the VM to another cloud server that satisfies the VM's security requirements.

3.1.3 Attestation Server

The Attestation Server acts as the attestation requester and appraiser, and consists of two essential modules. 1) The `Property Interpretation Module` is responsible for validating measurements, interpreting properties and making attestation decisions. It needs a certificate from a `privacy Certificate Authority` (pCA) to authenticate cloud servers. The `privacy Certificate Authority` may be a separate trusted server already used by the cloud provider for standard certification of public-key certificates that bind a public key to a given machine. 2) The `Property Certification Module` is responsible for issuing an attestation certificate for the properties monitored. There can be different Attestation Servers for different clusters of cloud servers, enabling scalability of the *CloudMonatt* architecture.

We introduce the Attestation Server for security monitoring while the Cloud Controller is responsible for management. This job split achieves better scalability, since different attestation servers can be added to handle different clusters of cloud servers. It consolidates property interpretation in the attestation servers, rather than replicating this in each cloud server, or burdening the Cloud Controller. This also achieves better "separation of duties" security, since the Cloud Controller need only focus on cloud management while the Attestation Server focuses on security. It also improves performance by preventing a bottleneck at the Cloud Controller if it had to handle management as well as myriad attestation requests and security property interpretations.

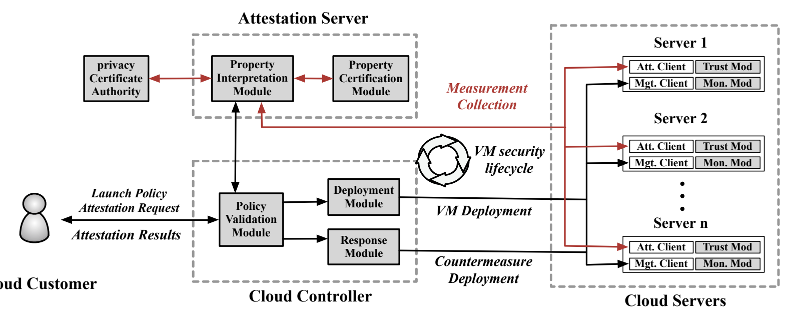
3.1.4 Cloud Server

The Cloud Server is the computer that runs the Virtual Machine (VM) in question. It is the attester in the system. It provides different measurements for different security properties. Figure 1b shows the structure of a cloud server with a Type-I hypervisor (e.g., Xen [41]). This has the hypervisor sitting on bare metal, and a privileged VM called the host VM (or Dom0) running over the hypervisor. Not all the cloud servers in the cloud provider's data center have to be trusted (almost all existing ones are not), only those servers on which security monitoring is necessary need to be secure. To support *CloudMonatt*'s goals, a cloud server must include a `Monitor Module` and a `Trust Module`.

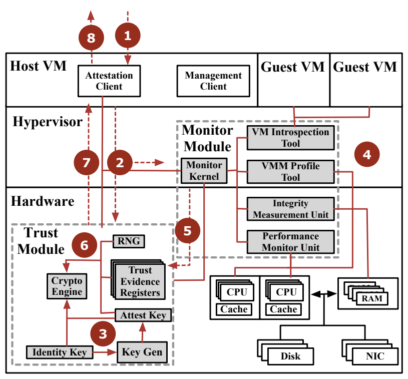
The `Monitor Module` contains different types of monitors to provide comprehensive and rich security measurements. These monitors can be software modules or existing hardware mechanisms like performance counters or the TPM chip. For example, the Performance Monitor Unit (present ubiquitously in Intel x86 [42] and ARM processors [43]) has numerous Hardware Performance Counters to collect runtime measurements of the VMs' activities. An Integrity Measurement Unit (which could use a TPM [10] chip) can be used to measure accumulated hashes of the system's code and static data configuration. In the hypervisor, a Virtual Machine Introspection tool (e.g., LibVMI [44]) can be used to collect the information inside the specified VM, and the VMM profile tool (e.g., xentrace [45]) can be used to collect dynamic information about each VM's activities.

We define a new hardware `Trust Module` in Figure 1b. This `Trust Module` is responsible for server authentication using the `Identity Key`, crypto operations using the `Crypto Engine`, `Key Generation` and `Random Number Generation` (RNG) blocks, and secure measurement storage using the `Trust Evidence Registers`. By using new hardware registers to store the security health measurements (trust evidence), we do not need to include the main DRAM memory in our Trusted Computing Base, although trusted RAM can also be used instead of `Trust Evidence Registers` in the Trust Module.

Figure 1b also shows the functional steps taken by the `Monitor Module` and the `Trust Module`. The Cloud Server includes an `Attestation Client` in the host VM



(a) Overview of the architecture



(b) Server architectures

Fig. 1: *CloudMonatt*

that ① takes requests from the Attestation Server to collect a set of measurements. It invokes the `Monitor Module` ② to collect the measurements and the `Trust Module` ③ to generate a new attestation key for this attestation session. This new attestation key is signed by the `Trust Module`'s private identity key. The required measurements of suspicious events or evidence of trustworthy operation are ④ collected from the `Monitor Module` and ⑤ stored into new `Trust Evidence Registers`. The `Trust Module` then ⑥ invokes its `Crypto Engine` to sign these measurements and ⑦ forwards the data to the `Attestation Client` which ⑧ sends it to the Attestation Server.

3.2 Threat Model

The threat model is that of hostile VMs running in the cloud on the same cloud server, or hostile applications or services running inside a VM, that try to breach the confidentiality or integrity of a victim VM's data or code. They may also try to breach its availability, in spite of the cloud provider having allocated the VM its requested resources. The cloud provider is assumed to be trusted (with its reputation at stake), but may have vulnerabilities in the system. We assume that the Cloud Controller and the Attestation Server are trusted - they are correctly implemented, with secure bootup and are protected during runtime. However the Cloud Servers need not be trusted, except for the `Trust Module` and `Monitor Module` in each server. Note that the Cloud Controller and Attestation Server can be redundancy protected for reliability and security, and are only a small percent of all the servers in the cloud's data center. Also, not all the thousands of cloud servers need to be *CloudMonatt*-secure servers.

We focus on two types of adversary's capabilities: (1) An adversary, who tries to exploit vulnerabilities in the customers' VMs, either from inside the VM, or from another malicious VM co-resident on the same server. (2) An active adversary who has full control of the network between different servers, as in the standard Dolev-Yao threat model [46]. The adversary is able to eavesdrop as well as falsify the attestation messages, trying to make the customer receive a forged attestation report without detecting anything suspicious. With regard to this second adversary, *CloudMonatt* needs secure monitoring and attestation protocols which we define in Section 3.4.

3.3 Security Health Monitoring

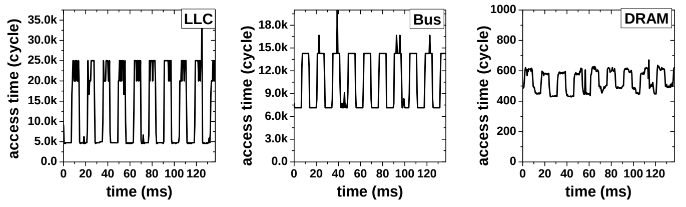
Different indicators of different aspects of security health can be monitored. In our context, these different aspects of security are the security properties requested by the customer. These security properties can be monitored by the various monitors in the server's `Monitor Module` or collected by the *Trust Evidence Registers* in the server's `Trust Module`. The *CloudMonatt* architecture is flexible and allows the integration of an arbitrary number of security properties and monitoring mechanisms, including logging, auditing and provenance mechanisms.

To monitor and attest a security property, three requirements must be satisfied: (1) the Attestation Server can translate the security property, requested for attestation by the customer, to the measurements to request from the target cloud server; (2) the target cloud server implements a `Monitor Module` that can collect these measurements, and a *Trust Module* with a `Crypto Engine` that can securely hash and sign the measurements and send them back to the Attestation Server. (3) the `Property Interpretation Module` in the Attestation Server is able to verify the measurements and auxiliary information, and interpret if the security property is satisfied.

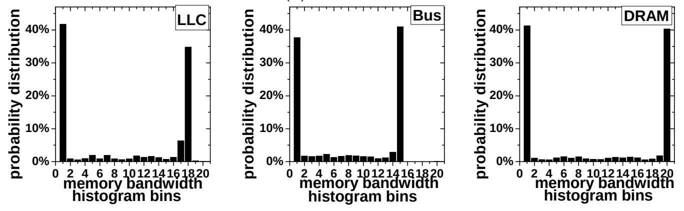
Property Mapping and Interpretation. The Attestation Server has a mapping of security property **P** to measurements **M**. This gives a list of measurements **M** that can indicate the security health with respect to the specified property **P**. The Attestation Server can also behave as the property interpreter and decision maker: when it receives the actual measurements **M'** from the server and VM, it can judge if the customers' requested security properties are being enforced.

Case Study. We give an example of detecting confidentiality vulnerabilities via covert channels to show an interesting use of *CloudMonatt*'s *Trust Evidence Registers*. The detection method is similar to that proposed in [47]. The real purpose of this example is to show how *CloudMonatt* can use *Trust Evidence Registers* to collect security measurements, e.g, to build an empirical probability distribution for attack detection.

Although VMs are isolated from each other by the hypervisor, it may still be possible to leak confidential data via a cross-VM covert channel at VM runtime. A covert channel exists when a colluding insider (e.g., a program inside the victim VM) can use a medium not normally used for communications to leak secret information to an



(a) Attacks



(b) Detection

Fig. 2: Frequency distribution detection of three covert channels.

unauthorized party in another VM. When VMs on the same server share physical resources, the contention for these shared resources can be exploited to encode and transmit information, e.g., in the form of timing features. Such characteristics can be different cache operations (hit or miss), memory bus activities (locked or unlocked bus), or DRAM controller states (bandwidth saturated or not). Figure 2a shows the covert channel information observed by the receiver VMs, using each of the Last Level Cache (LLC), bus and DRAM as the covert channel communication medium.

A key idea to detect these covert channels is that *programs involved in covert channel communications give unique patterns of the events happening on these hardware* [47]. If a customer requests covert channel protection and periodic attestation of this, *CloudMonatt* can use hardware performance counters to monitor the attested VM's memory bandwidth every 0.1ms. After a certain monitoring period, *CloudMonatt* calculates the frequency distribution histogram for the memory bandwidth used. Specifically, it divides the entire range of observed bandwidth values into 20 bins with equal size, and then counts how many bandwidth values fall into each bin. Then *CloudMonatt* uses 20 *Trust Evidence Registers* to store the number of values in each bin to represent the memory bandwidth distribution. These 20 values are sent as the security health measurements for detecting these LLC, bus or DRAM covert channels. We use 20 bins in our experiment, but a different number can be used to save space or increase accuracy.

When the Attestation Server receives the 20 values, the `Property Interpretation Module` calculates the probability distribution (Figure 2b) of the memory bandwidth. If a covert channel exists, the distribution graph gives two peaks: each peak representing the activity of transmitting a "0" or a "1", respectively. The Attestation Server can use machine learning techniques to conduct pattern recognition of covert channels. More sophisticated detection methods can be integrated into *CloudMonatt* to detect other types of attacks.

3.4 Monitoring and Attestation Protocols

In a distributed architecture where communication is over untrusted networks, the protocols are an essential part of the security architecture: they establish trust between the customer and the cloud provider, and between different computers in the cloud system. In *CloudMonatt*, an attestation protocol must be unforgeable in spite of the network attacker and the other attackers in the untrusted servers. This requires secure communications among the four entities in Figure 1a, and unforgeable signatures of the measurements and the attestation report from the place of collection (in the Cloud Server) through the Attestation Server, Cloud Controller and finally to the customer. We first describe the main attestation protocol. Details of the cryptographic keys involved, the secure communications and storage will be clarified later.

Figure 3 shows the attestation protocol in *CloudMonatt*.

1) The customer initially sends to the Cloud Controller the attestation requests including the VM identifier **Vid**, the desired security property **P** and a nonce $\mathbf{N}_1$. The nonce is an arbitrary number used only once in this session. It is used to prevent replay attacks over the channel between the customer and the Cloud Controller.
2) The Cloud Controller knows the mapping of all VMs to their assigned cloud servers. It discovers the host server of VM **Vid**, **I**, and sends to the Attestation Server the request, which includes **Vid**, **I**, **P** and another nonce $\mathbf{N}_2$.
3) Given the property **P**, the Attestation Server identifies the required monitoring measurements **rM**. Then it sends **Vid**, **rM** and its nonce $\mathbf{N}_3$ to the cloud server **I** where the VM is running.
4) In the Cloud Server, the `Monitor Module` collects the required measurements **M** and stores them into the `Trust Evidence Registers`. Then the `Trust Module` calculates the quote $\mathbf{Q}_3$ as the hash value of (**Vid**, **rM**, **M** and nonce $\mathbf{N}_3$) (We borrow the term "Quote" from TPM notation, to represent a cumulative hash measurement), and sends to the Attestation Server a signature of **Vid**, **rM**, **M**, $\mathbf{N}_3$ and $\mathbf{Q}_3$.
5) The Attestation Server verifies the signature and checks the integrity of the measurements by calculating the hash value and comparing it with the quote $\mathbf{Q}_3$. Then it interprets the measurements **M** and property **P** and generates the attestation report **R**. The Attestation Server calculates the quote $\mathbf{Q}_2$ as the hash value of (**Vid**, **I**, **P**, **R** and $\mathbf{N}_2$), and sends to the Cloud Controller a signature of **Vid**, **I**, **P**, **R**, $\mathbf{N}_2$ and $\mathbf{Q}_2$.
6) The Cloud Controller verifies the signature and checks the integrity of the report **R** via the hash value $\mathbf{Q}_2$. Then it generates the quote $\mathbf{Q}_1$ by hashing **Vid**, **P**, **R** and $\mathbf{N}_1$, signs these values and sends the signature to the customer.
7) The customer verifies the signature and hash value. If they are correct, the customer gets the correct report **R**.

3.4.1 Secure Storage and Communications:

For secure storage, the `Trust Module` provides `Trust Evidence Registers` for saving attestation measurements, which are only accessible to the `Trust Module` and `Monitor Module`. Accesses to the databases in the Cloud Controller and the Attestation Server are also protected to ensure data confidentiality and integrity.

For secure communications over networks, the *CloudMonatt* architecture expects the customer, Cloud Controller, Attestation Server and secure Cloud Servers to implement the SSL protocol. Our contribution is defining the *contents* of the SSL messages, and the keys and signatures required for unforgeable attestation reports and Cloud Server anonymity.

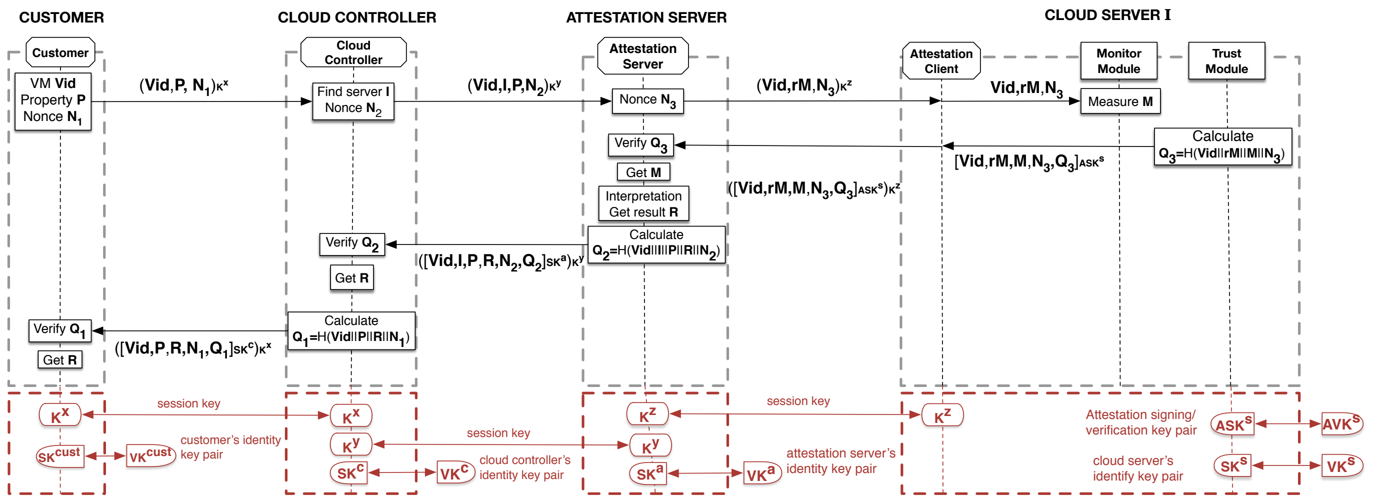


Fig. 3: Attestation Protocol and Key Management in *CloudMonatt*. We use the notation $[M]_K$ for a private key operation with key K, $\{M\}_K$ for a public key operation with key K, and $(M)_K$ for a symmetric key operation with symmetric key K. N_i represents a Nonce between two communication parties.

3.4.2 Key Management

We now describe the keys used in Figure 3. The Cloud Controller, Attestation Server and each secure Cloud Server must have one long-term public-private key-pair that uniquely identifies it within the cloud system. This is minimally what is required for SSL support, and is already present in all cloud servers. Hence, each secure cloud server owns a pair of public-private identity keys, $\{\mathbf{VK}^s, \mathbf{SK}^s\}$. The private key, $\mathbf{SK}^s$, can be burned into the `Trust Module` when manufactured, or more preferably, securely inserted into a non-volatile and tamper-proof register in the `Trust Module` when the server is first deployed in the cloud data center. This private identity key is never released outside of the `Trust Module`. The public key, $\mathbf{VK}^s$, can be used to authenticate the cloud server. A cloud server mainly uses this identity key-pair to generate a temporary key pair for each attestation request.

A new session-specific key-pair, $\{\mathbf{AVK}^s, \mathbf{ASK}^s\}$, is created by the `Trust Module` whenever an attestation report is needed, so as not to reveal the location of a VM. The public attestation key $\mathbf{AVK}^s$ is signed by the Cloud Server's $\mathbf{SK}^s$ and sent to the pCA for certification. The pCA verifies the signature via $\mathbf{VK}^s$ and issues the certificate for $\mathbf{AVK}^s$ for that server. This certificate enables the Attestation Server to authenticate the server "anonymously" for this attestation.

For secure communications between the servers, SSL first authenticates sender and receiver using their public-private key-pairs, then generates symmetric session keys for encrypting the messages passed between each pair of servers. Hence, Figure 3 shows the communications between the customer and the Controller protected with a symmetric key $\mathbf{K}^x$, between the Controller and the Attestation Server with a symmetric key $\mathbf{K}^y$, and between the Attestation Server and Cloud Server with a symmetric key $\mathbf{K}^z$.

4 Implementation

We implemented our property-based cloud attestation on the OpenStack platform [48]. We integrated the OpenAttestation software (oat) [49] for host remote attestation protocols. We integrated the TPM-emulator [50] and leveraged it to emulate the functions of the `Trust Module` in the hardware. We make *CloudMonatt* open-source and available online [1]. Figure 4 displays our prototype implementation.

4.1 Cloud Controller

The Cloud Controller is implemented by modifying OpenStack. OpenStack is composed of different services. We modified two services. The first one is *horizon*, which is implemented as OpenStack's dashboard and provides a web-based user interface to customers. The second one is *nova*, which is used to manage computing services in cloud servers.

We modify four OpenStack modules to implement the Cloud Controller: (1) *horizon*: we extend the VM launch interface with the monitoring and attestation options: when launching VMs, the customers can specify which properties they want for their VMs. We also enable the customers to start or disable the security health monitoring during the VMs' lifetime. (2) *nova api*: we modify this module to pass new VM launch options, monitoring requests from *horizon* to *nova*, as well as attestation results from *nova* to *horizon*. (3) *nova database*: we modify the controller's database to enable it to store the customers' specifications about the security properties required for their VMs, from *nova api*. We also add new tables in the database, which record each servers' monitoring and attestation capabilities: i.e., what properties they support for monitoring. (4) *nova scheduler*: we modify this to implement the `Policy Validation Module` and `Deployment Module` of the Cloud Controller. It is responsible for choosing the host for the VM during initial allocation and migration. We add a new filter: *property_filter*, to select qualified servers to host VMs based on their customers' security properties, monitoring and attestation requirements.

We add two new modules in the controller: (1) *nova attest_service:* this module manages the attestation services. It connects *nova database* (for retrieving security properties), *oat api* (for issuing attestations and receiving results) and *nova response* (for triggering the responses). (2) *nova response:* this implements the `Response Module`, responsible for providing some responses if the attestation fails.

1. https://github.com/eepalms/CloudMonatt.git

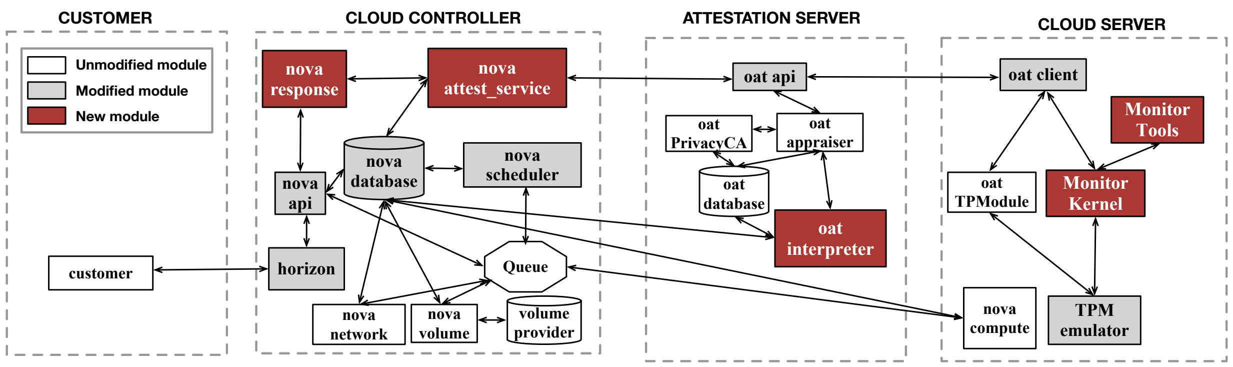


Fig. 4: Implementation of *CloudMonatt* Architecture.

4.2 Attestation Server

The attestation server and client are realized by OpenAttestation. Three of the four main modules remain unchanged: *oat database* stores information about the cloud servers and measurements; *oat appraiser* is responsible for triggering attestations and reporting the measurements; *oat PrivacyCA* provides public-key certificates for the cloud servers.

We modify *oat api* by extending the APIs with more parameters, i.e., security properties and VM id. We add a new module *oat interpreter*: this essential new module implements the Property Interpretation and Certification Modules of the Attestation Server. It can interpret the security health of the VM and make attestation decisions, based on the information of the cloud server from the *nova database* and the security measurements from the *oat database*.

4.3 Cloud Servers

In each cloud server, *nova compute* is the client side of OpenStack nova. We modify *oat client*, the client side of OpenAttestation, to receive attestation requests. We modify the *TPM emulator* to provide secure storage and crypto functions. We add two new modules: *Monitor Kernel* can start the security measurements and store the values into the *TPM emulator*, and `Monitor tools` can integrate different software VMI tools, VMM Profile tools or other logging or provenance tools, into the server to perform the monitoring and take measurements.

5 Security Verification

We conduct a security verification of *CloudMonatt*. We aim to address two questions: (1) can *CloudMonatt* provide unforgeable VM health reports to customers and the cloud provider? (2) What are the minimal security requirements (i.e., software/hardware modules that need to be trusted) that can guarantee the security and correctness of *CloudMonatt*?

5.1 Verification Methodology

To verify a system's protocols and operations, we first specify the verification goals and invariants based on the system's functionality. Then we build models for the system, and identify the trusted and untrusted subjects in the system. We implement the models and verification invariants in a cryptography verification tool and run this tool to test if the invariants pass for every possible path through the system models from the initial state to the end state. If an invariant fails in some cases, we try to find the vulnerabilities and construct the corresponding attacks. We describe these steps for verifying *CloudMonatt* in detail below.

Analyzing verification goals. *CloudMonatt* has two basic functionalities: (1) reporting VMs' potential security threats to the cloud provider so it can take the corresponding countermeasure to mitigate the threats; (2) notifying the customers of their VMs' security health. So *CloudMonatt* must ensure that the cloud provider and customers can receive the correct and unforgeable monitoring reports. These are the two verification goals of *CloudMonatt*.

Figure 5 shows the structure of verification goals and their dependent conditions. The two red blocks (at the top and left) show the two main goals we want to verify: (1) the goal that the customers can receive the correct reports depends on three conditions: the Cloud Controller can receive the correct reports, process them correctly and transmit them securely to the customers. (2) The goal that the Cloud Controller can receive the correct reports also depends on three conditions: the Attestation Server can receive the correct measurements, process them (i.e., generate correct reports) correctly, and transmit the reports to the Cloud Controller securely. In addition to the above two main goals, the condition that the Attestation Server can receive the correct measurements depends on two conditions: the cloud server collects correct measurements, and such measurements can be transmitted to the Attestation Server securely. The trustworthiness of each server depends on two conditions: the critical software and hardware modules function correctly, and messages are exchanged securely between these modules.

In order to verify the main goals in a scalable way, we break the verification task into two steps, adapting and extending the methodology from [7]. The first step is *external verification*, which aims to verify the main verification goals (red blocks in Figure 5). In this step, we treat each server as a blackbox (dashed boxes in Figure 5). For each server we only consider the black block as a precondition and assume it is already satisfied, while ignoring other basic preconditions (grey blocks) inside the boxes. Under such preconditions and other basic preconditions outside of the blackboxes, we verify if the main goals are held. The second step is *internal verification* in which we consider the activities inside each server. In this step, the precondition we assume in the previous step becomes the postcondition that we want to verify. We want to check if such postcondition is held, i.e., the precondition we make in the previous step is correct, under the basic preconditions inside the dashed box.

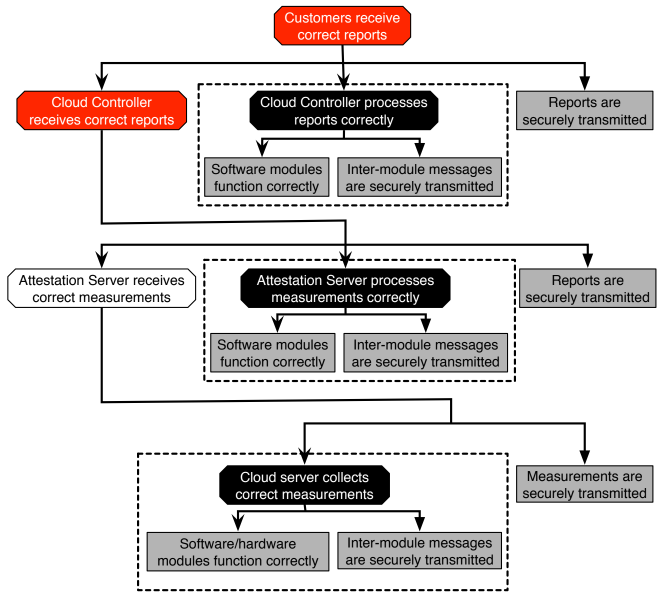

Customers receive correct reports
Cloud Controller receives correct reports
Cloud Controller processes reports correctly
Reports are securely transmitted
Software modules function correctly
Inter-module messages are securely transmitted
Attestation Server receives correct measurements
Attestation Server processes measurements correctly
Reports are securely transmitted
Software modules function correctly
Inter-module messages are securely transmitted
Cloud server collects correct measurements
Measurements are securely transmitted
Software/hardware modules function correctly
Inter-module messages are securely transmitted

Fig. 5: The structure of verification goals of *CloudMonatt*. Red blocks are the main goals of *external verification*. Black blocks are the preconditions of *external verification*, as well as the postconditions of *internal verification*. Grey blocks are the basic preconditions.

Modeling systems. To verify the above goals of *CloudMonatt*, we need to translate the system protocols and the underlying architectures into representative yet tractable models. We adopt the symbolic modeling method [51], where the cryptographic primitives are represented by function symbols and perfect cryptography is assumed. Specifically, we first specify subjects involved in this verification procedure. A subject can be a customer or a server in the distributed system, or a hardware/software module inside a server. For *external verification*, since we treat each server as a blackbox, we model each server and the customer as a subject. For *internal verification*, we need to consider the internal activities inside the server, so we model each software and hardware module involved in the system operation as a subject. Each subject has a set of states with inputs and outputs based on the system operation. The transitions between different states are also defined by the architecture designs and protocols.

Among all the subjects, there is an initiator subject that starts the system protocol and a finisher subject that ends the protocol. (The initiator and finisher could be the same subject). This initiator subject has a "Start" state while the finisher subject has a "Done" state. The verification procedure starts at the initiator's "Start" state. At each state in each subject, it takes actions corresponding to the transition rules. It will exhaustively explore all possible rules and states to find all the possible paths from the initiator's "Start" state to the finisher's "Done" state. Then we judge if the verification goals are satisfied in all of these paths. The system is verified to be secure if *there are paths from initiator's "Start" state to finisher's "Done" state, and all the verification goals are satisfied in any of these paths.*

Specifying security invariants. Invariants are conditions that need to hold true for there to be no violation of the verification goals or postconditions. The invariants can be specified from the goals or postconditions that we want to verify. For *CloudMonatt*, the goals of *external verification* are to ensure the customer and the Cloud Controller receive the correct reports. So the invariants are that *the reports received by the customer and the Cloud Controller are always the ones matching the security property and VM id they specify*. The postconditions for *internal verification* are to ensure that the servers process the data correctly. So the invariants are that *the output (e.g., measurements, report) sent from the server are always the ones correctly mapped to the input sent to the server*.

Identifying preconditions. Preconditions refer to the basic requirements that are needed to keep the security invariants true within the system protocols or operations. Basically it specifies the necessary subjects (e.g., network links connecting different servers, software or hardware modules inside the server) that should be trusted. For *external verification*, the preconditions are the assumptions we make about each server. For *internal verification*, the preconditions are the subjects that should be included in the Trusted Computing Base. The verification results can help us identify the minimal TCB for *CloudMonatt*, i.e., the necessary and critical software/hardware modules or servers that should be well protected to guarantee the correctness of *CloudMonatt*.

In the next two sections, we conduct the *external verification* and *internal verification* separately. We use *ProVerif* [8] to model the system and verify the security invariants. *ProVerif* is a software tool for checking security properties in cryptographic protocols. It supports a variety of cryptographic primitives, e.g., symmetric and asymmetric cryptography, digital signatures, hash functions, etc. If a security property is proven unsatisfied, *ProVerif* can reconstruct the attacker execution trace that falsifies the property. We show how to use *ProVerif* to check the system interactions, in addition to network protocols.

5.2 External Verification

Modeling. We model each server involved in this distributed system as an interacting state machine, as shown in Figure 6. Each subject is made up of some states. The customer is the initiator as well as the finisher subject. The whole process starts from the customer side, who sends to the Cloud Controller the attestation request including the VM identifier **Vid** and the desired security properties **P**. Then the Cloud Controller discovers the host cloud server, and forwards the request to the Attestation Server, with the server identifier **I**. The Attestation Server identifies the necessary monitoring measurements and sends the measurement request **rM** to the host cloud server. The cloud server collects the required measurements, calculates the hash value, **Q**, of the measurements requested and then sends these values back to the Attestation Server, after which the cloud server reaches the "Done" state. The Attestation Server checks the signature, the hash value and the nonce: if this check fails, the Attestation Server goes to "Abort" state. Otherwise it interprets the measurements and the property, and generates the attestation report **R**, as explained in Section 3.4. Then the Attestation Server signs the report, transmits it to the Cloud Controller, and goes to state "Done". After receiving the report, the Cloud Controller checks the signature, the hash value and the nonce. If anything is incorrect, the Cloud Controller goes to state "Abort". Otherwise it hashes and signs the report, and ends at state "Done" after sending the report to the customer. If the customer finds the encrypted

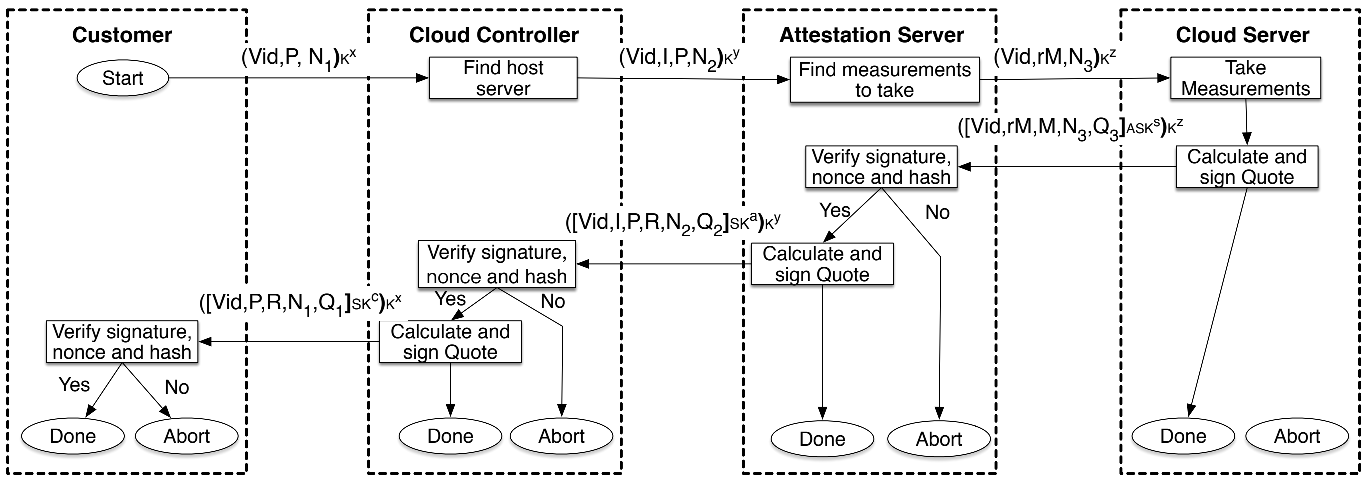


Fig. 6: The external protocol in *CloudMonatt*. $\mathbf{K}^x$, $\mathbf{K}^y$ and $\mathbf{K}^z$ are symmetric keys between the customer and the Cloud Controller, between the Cloud Controller and the Attestation Server, and between the Attestation Server and the cloud server, respectively. $\mathbf{SK}^c$, $\mathbf{SK}^a$ and $\mathbf{ASK}^s$ are the private signing keys of the Cloud Controller, the Attestation Server and the cloud server, respectively. $\mathbf{N}_1$, $\mathbf{N}_2$ and $\mathbf{N}_3$ are the nonces used by the customer, the Cloud Controller and the Attestation Server, respectively

signature of the report is correct, it goes to state "Done". Otherwise, it goes to state "Abort".

Security invariants. As we discussed in Section 5.1, the external verification is to check if the customer and cloud provider can receive the correct attestation reports. We identify several specific security invariants for this task in our modeled state machines:

① The Cloud Controller is able to reach state "Done". When it is at state "Done", the attestation report **R** it receives is indeed the one for VM **Vid** with property **P**, specified by the customer.

② The customer is able to reach state "Done". When he is at state "Done", the attestation report **R** he receives is indeed the one for VM **Vid** with property **P**, specified by the customer.

Invariant ① is to ensure the Cloud Controller gets the correct attestation reports. Invariant ② is to ensure the customer gets the correct attestation reports.

Preconditions. We make several preconditions about each server and check if the above security invariants can be satisfied under these preconditions. These preconditions indicate the subjects that should be included in the TCB. Verifying the sufficiency and necessity of these preconditions can help us find the minimal TCB for *CloudMonatt*.

(C1) The cloud server is trusted.
(C2) The Attestation Server is trusted.
(C3) The Cloud Controller is trusted

Here a "trusted" server means it will strictly follow the operations indicated in our protocol. For instance, a trusted cloud server will collect and sign correct measurements; a trusted Attestation Server will process the measurements and generate the reports correctly; a trusted Cloud Controller will process the VM health reports correctly. In addition, a trusted server will keep its secrets from attackers.

Implementation. We model the authentication and communication protocols of *external verification* in *ProVerif*. Specifically, we declare each subject (the customer, the Cloud Controller, the Attestation Server and the cloud server) as a process. Inside the process we model the operations of state machines shown in Figure 6. Each process keeps some secrets (e.g., cryptographic keys, attestation reports or measurements). If the subject is trusted, then the attacker cannot get these secrets, and we use the keyword `private` to denote these variables. Otherwise the variables are declared as public and the attacker can obtain the values.

To model the network activities in this system, we declare a `channel` between each pair of subjects, to represent the untrusted communication channel. These channels are under full control of the network-level adversaries, who can eavesdrop or modify messages transmitted in the channels.

We also use the cryptographic primitives from *ProVerif* to model the public key infrastructure for digital certificate, authentication and key exchange. Then we model all the steps in Figure 6 for an unbounded number of attestation sessions, i.e., the customer keeps sending attestation requests to the cloud system and receiving the reports. *ProVerif* can check if the adversary can compromise the integrity of the report in any attestation session, and display the attack execution trace if a vulnerability is found.

We can use *ProVerif*'s reachability proof functionality to verify if the Cloud Controller and customer are each able to reach state "Done". *ProVerif* allows us to define an event **E** inside a process at one state, which specifies some conditions. Then we can check if this event will happen when the protocol proceeds using the query statement: "`query event`(**E**)". *ProVerif* can enumerate all the possible execution traces and check if this event is reachable in some cases. If so, this query statement returns true as well as the trace that reaches the event. Otherwise the statement returns false. So we can use the statement "`query event`(**Done**)" to check if the customer and Cloud Controller can receive the report.

ProVerif does not provide direct functionalities to prove integrity. However, we can also use its reachability proof functionality to verify the integrity property of a message. Specifically, to verify the integrity of the attestation report in invariant ①, we check if the report received by the Cloud Controller, **R′** is the correct one, **R**, determined by the VM identifier **Vid**, the security property **P** and the VM's state measurement **M**, when the Cloud Controller reaches state "Done". Then we establish an event: "($\mathbf{R'} \neq \mathbf{R}$)" at state "Done" to denote the integrity breach. We use the statement "`query` `event`(**R′**≠**R**)" to verify the integrity. If this statement is false, it means the attacker has no means to change the message **R** without being observed by the Cloud Controller. Then the integrity of **R** holds. Similarly, to verify invariant ②, we check if the report **R′** received by the customer at state "Done", is the correct one **R**, by checking if the statement "(**R′**≠**R**)" at state "Done"is false.

Results. *Proverif* shows that state "Done" is reachable for both the Customer and Cloud Controller. Then we verify if the security invariants ① and ② are satisfied under the preconditions (C1) – (C3). First, *ProVerif* confirms that preconditions (C1) – (C3) are sufficient to guarantee that the customer and the Cloud Controller can receive the correct attestation reports. Note that as we put trust on the Cloud Controller, the Attestation Server and the cloud server, we do not need to consider the server-level adversaries. Even though the network-level adversaries can take control of all the network channels between each server, they cannot compromise the integrity of the messages without being observed, since all the messages are hashed, signed and encrypted before being sent to the network.

Second, we check if preconditions (C1) – (C3) are necessary to keep the invariants correct. *ProVerif* shows that it is necessary to place the subjects of (C1), (C2) and (C3) in the TCB. Missing any precondition can lead to violations of some invariants: if the cloud server is not trusted, then the server-level adversary can counterfeit wrong measurements, causing the Attestation Server to make wrong attestation decisions, and pass them to the Cloud Controller and the customer. So invariants ① and ② are not satisfied. If the Attestation Server is not trusted, then it can generate wrong attestation reports for the customer and the Cloud Controller. So invariants ① and ② are not satisfied. If the Cloud Controller is untrusted, it can modify the reports before sending to the customer. So invariant ② is not satisfied.

In the next section, we perform *internal verification* of the trusted servers, to show which component in each of the servers should be trusted, in order to satisfy the preconditions we assume in this section.

5.3 Internal Verification

From Section 5.2 we know that to satisfy the external verification goals, we need to assume the correctness of the preconditions in each server, i.e., trusting the data processing in the Cloud Controller, the Attestation Server and the cloud server. However, we do not need to place the entire server into the TCB. On the one hand, trusting each component in one server is not a necessary condition to satisfy the precondition we assume for this server. Also, including all the components of the server into the TCB would require stronger security protection for the entire server, which is expensive and difficult to achieve. On the other hand, it is impossible to trust every component in the server, especially for the cloud server which hosts the guest VMs rented to the customers. *CloudMonatt* cannot ensure that the guest VMs are trusted. As such, we conduct the *internal verification* to identify which components inside the server need to be trusted, to satisfy the preconditions in the *external verification*.

5.3.1 Cloud Server

First, we verify the system interactions on the cloud server.

Modeling. We abstract the key components inside a cloud server, and model them as state machines, as shown in Figure 7. We also include the Attestation Server to interact with the cloud server. The Attestation Server is the initiator and finisher subject in the internal protocol. The whole process starts when the Attestation Server sends the measurement request to the cloud server. The `Attestation Client` processes the request and passes it to the `Monitor Module`. The `Monitor Kernel` inside the `Monitor Module` figures out the corresponding monitor tool and invokes it to collect the correct measurements. Then it stores the measurements together with other related information in the `Trust Evidence Registers`. Then the `Crypto Engine` in the `Trust Module` retrieves the measurements, calculates the quote (see Section 3.4) and signs it using the `Attestation Key`. Then the signature is encrypted by the `Attestation Client` and sent to the Attestation Server. After this all the subjects inside the cloud server reach the state "Done". The Attestation Server will check the hash and signature. It goes to state "Done" if the check succeeds and state "Abort" if the check fails.

Security invariants. the invariant for *internal verification* is to check if the cloud server collects the correct measurements and sends them to the Attestation Server. So we translate this invariant to the statement as below:

① The Attestation Server is able to reach state "Done". When it is at state "Done", the measurements **M** it receives is indeed the one for VM **Vid** with request **rM**, which were sent to the cloud server.

Preconditions. We identify a set of possible preconditions to satisfy the above invariant. We classify these preconditions as different modules and inter-module communications. We check the necessity and sufficiency of these preconditions for guaranteeing the integrity of measurements taken from the cloud server.

1. `Attestation Client`:
   - (C1.1) this module is trusted.
2. `Monitor Module`:
   - (C2.1) the `Monitor Kernel` is trusted.
   - (C2.2) the `Monitor Tools` are trusted.
   - (C2.3) the channel between the `Monitor Kernel` and the `Monitor Tools` is trusted.
3. `Trust Module`:
   - (C3.1) the `Crypto Engine` is trusted.
   - (C3.2) the `Trust Evidence Registers` are trusted.
   - (C3.3) the `Attestation Key` is securely stored.
   - (C3.4) the channel between the `Attestation Key` and the `Crypto Engine` is trusted.
   - (C3.5) The channel between the `Crypto Engine` and the `Trust Evidence Registers` is trusted.
4. Inter-module communication:
   - (C4.1) the channel between the `Monitor Kernel` and the `Attestation Client` is trusted.
   - (C4.2) The channel between the `Attestation Client` and the `Crypto Engine` is trusted.
   - (C4.3) The channel between the `Monitor Kernel` and the `Trust Evidence Registers` is trusted.

Implementation. *ProVerif* does not provide functionalities for modeling and verification of architecture-level interactions. However, we can model the server system as a network

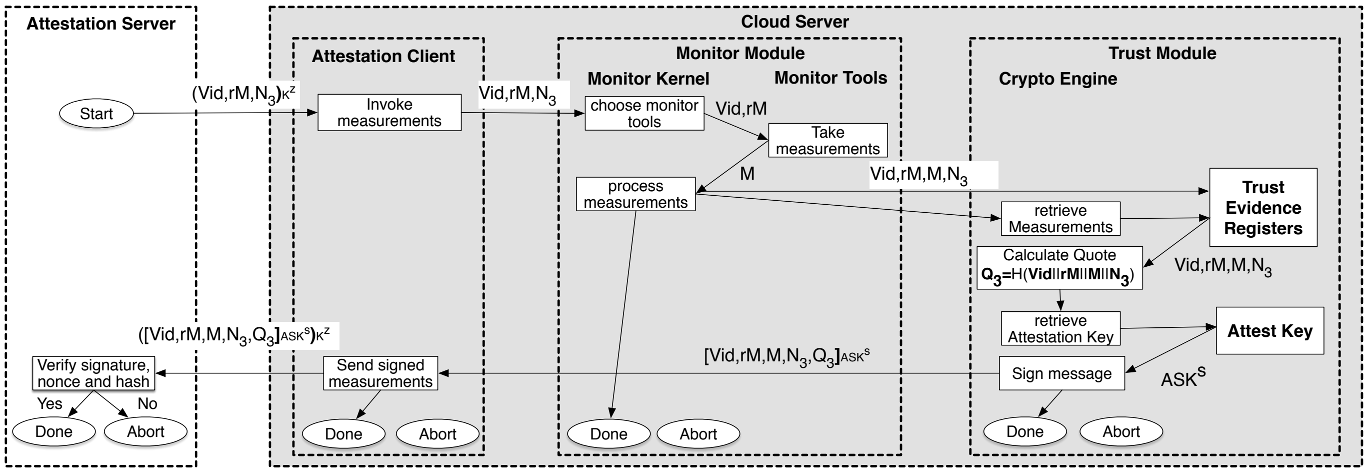


Fig. 7: Internal protocol (interactions) in the cloud server

system, and verify the server in a similar way as the network protocol verification. Specifically, we can model a software or hardware component as a process. Each component keeps some variables and operates as a state machine. If one component is in the TCB, then its variables will be declared as `private`. Otherwise its variables are public to attackers. If the attacker has the privilege to control the communication between two components, then we declare a public `channel` for these two components. On the contrary, if two modules are linked by one channel that is trusted, then we combine the two processes into one process so that the two modules can exchange messages directly without being compromised by third party attackers.

We model all the steps in Figure 7 for an unbounded number of sessions, i.e., the Attestation Server keeps sending measurement requests to the cloud server and receiving the results. *ProVerif* enumerates all the possible states during the infinite sessions and checks if the property is maintained.

Results. *ProVerif* shows that state "Done" is reachable for the Attestation Server. Then, we consider and verify the sufficiency and necessity of the above preconditions that satisfy the security invariants. We use the same reachability functionality of *ProVerif* to verify the integrity property under different preconditions.

For precondition (C1.1), *ProVerif* shows that the integrity property is still satisfied, and the adversary cannot tamper with the messages, even if he takes control of the `Attestation Client`. If the adversary changes **Vid** or **rM** before they are sent to the `Monitor Module`, the `Monitor Module` will collect the wrong measurements **M**. However, the `Trust Module` will also sign the modified **Vid** or **rM**. The Attestation Server will notice this modification and go to state "Abort". So (C1.1) is not a necessary condition and can be removed from the TCB.

For (C2.1), (C2.2) and (C2.3), *ProVerif* shows that without any of the three preconditions, the integrity checking of measurements will fail. *ProVerif* also shows attack execution traces if one precondition is missing. For instance, if the `Monitor Kernel` is untrusted, it can send a different **Vid** or **rM** to the `Monitor Tools` to collect wrong measurements **M**. If the `Monitor tools` are untrusted, even if they receive the correct measurement request, they can give wrong measurement data. If the communication channel is open to the adversary, he can easily modify the measurement requests or results without being noticed by the other trusted subjects. So (C2.1), (C2.2) and (C2.3) are necessary conditions to protect the measurements' integrity, and must be kept.

ProVerif shows preconditions (C3.1) – (C3.5) are also necessary to guarantee the integrity of measurements. It also shows the attack execution traces without these conditions. If the attacker compromises the `Crypto Engine`, the `Attestation Key` or their communication channel, it can generate a fake signature over any measurements using the signing key $\mathbf{ASK}^S$, while the Attestation Server will never detect this integrity breach. If the `Trust Evidence Registers` or their connection with the `Crypto Engine` are compromised, then a server-level adversary can easily tamper with the security measurements stored in the untrusted `Trust Evidence Registers` or transmitted in the untrusted channel, without being detected by the Attestation Server.

Precondition (C4.1) is not necessary, with the same reason as precondition (C1.1). Precondition (C4.2) is not necessary. The adversary cannot compromise the message integrity since the message in this channel is signed. Precondition (C4.3) is necessary. If this channel is not trusted, the adversary can modify the measurements, **M**, then the `Trust Module` will store and sign the wrong measurements.

Based on the above results, the necessary conditions to guarantee the measurements' integrity are: (1) the `Monitor Module` is trusted (i.e., the `Monitor Kernel`, the `Monitor Tools`, and their communications); (2) the `Trust Module` is trusted (i.e., the `Crypto Engine`, the `Attestation Key`, the `Trust Evidence Registers`, and their communications); (3) the communication channel between the `Monitor Module` and the `Trust Module` is trusted. *ProVerif* shows that it is also sufficient for the cloud server to maintain the property of measurements' integrity if only these subjects are included in the TCB of a cloud server. *In particular, ProVerif shows that the* `Attestation Client` *need not be trusted.*

5.3.2 Attestation Server

We use the same method to verify the Attestation Server.

Modeling. Figure 8 shows the state machines of the Attestation Server. The Attestation Server has two modules: the `Property Interpretation Module` and the `Property Certification Module`. The `Property`

`Interpretation Module` is used to invoke the attestation. The `Property Certification Module` is used to provide the measurements certification for one security property.

Security invariants. The invariant for verification of the Attestation Server is to check if the Attestation Server generates the correct attestation report and sends it to the Cloud Controller:

① The Cloud Controller is able to reach state "Done". When it is at state "Done", the report **R** it receives is indeed the one for VM **Vid** with property **P**, which were sent to the Attestation Server.

Preconditions. We identify a set of preconditions for the Attestation Server:

(C1) The `Property Interpretation Module` is trusted.
(C2) The `Property Certification Module` is trusted.
(C3) The channel between the `Property Certification Module` and `Property Interpretation Module` is trusted.

Implementation. We use *ProVerif* to verify the Attestation Server in a similar way as the cloud server.

Results. We find that both of the two modules and their communications should be placed into the TCB of *CloudMonatt*. If the `Property Certification Module` is incorrect, it will give wrong property certifications. If the `Property Interpretation Module` is compromised, it can invoke the wrong attestation requests or send wrong attestation reports. Untrusted channels between these modules will bring the same attack effects.

5.3.3 Cloud Controller

Modeling. In the Cloud Controller, we consider the interactions between three modules: the `Policy Validation Module`, `Deployment Module` and `Response Module`. The state machines of these three modules are displayed in Figure 9. We consider two events: the customer launches the VM, and issues runtime attestation requests.

Security invariants. The invariant for verification of the Cloud Controller is to check if the Cloud Controller can send the correct attestation report to the cloud customer.

① The customer is able to reach state "Done". When he is at state "Done", the report **R** he receives is indeed the one for VM **Vid** with property **P**, specified by him.

Preconditions. The preconditions for the Cloud Controller are related to the three modules involved:

(C1) The `Policy Validation Module` is trusted.
(C2) The `Deployment Module` is trusted.
(C3) The `Response Module` is trusted.
(C4) The channel between the `Policy Validation Module` and `Deployment Module` is trusted.
(C5) The channel between the `Policy Validation Module` and `Response Module` is trusted.

Implementation. The verification of the above models is similar to the cloud server and the Attestation Server.

Results. To guarantee the integrity of attestation reports, the `Policy Validation Module` must be trusted. If this module is compromised, then the whole monitoring service will be compromised. For the `Deployment Module` and the `Response Module` and their communication channels with the `Policy Validation Module`, they are not necessary to protect the integrity of attestation reports. However, they are used to control VMs. So they should also be trusted to guarantee that the cloud system functions correctly.

Model	**Int. or Ext.**	**Lines of Code**	**Run--time**
External	Ext.	262	0.2
Cloud Server	Int.	123	0.1
Attestation Server	Int.	205	0.2
Cloud Controller	Int.	187	0.1

TABLE 1: Verification evaluation results. First column shows what is being modeled, second column shows if it is internal interaction (Int.) or external protocol (Ext), third column shows the lines of code, and last column shows the run time (in seconds).

5.4 Verification Evaluation

We measure the verification effort for the above cases. Table 1 shows what is being modeled, followed by the lines of code and the run time (in seconds). The lines of code include some comments which are very helpful for understanding the verification. The verification process is iterative, where the ProVerif files may be updated many times, thus comments are crucial to understand the development of the verification strategy. The code has not been optimized for minimal size, nevertheless, the size is very small and runtime is very short for all the *ProVerif* verification. The run time for verification is also very small: due to the breakdown into internal and external verification, we can verify complex architectures within a very short time. The most effort-consuming step is the design and writing of the verification models, but the actual verification is very fast.

5.5 Verification Implication

The external and internal verification results can help us verify and enhance the security of *CloudMonatt*. We identify the components that are required in *CloudMonatt*'s TCB. Then we can use existing software-hardware solutions to protect these components.

5.5.1 Cloud Server Protection

Verification results show that the `Monitor Module` and `Trust Module` of a cloud server should be included in the TCB. Normally, third party customers only get guest VM privilege (ring 0) while the `Monitor Module` and `Trust Module` have hypervisor privilege (ring -1). So a normal tenant has no capability to subvert the security functions provided by these two modules. To enhance the protection of *CloudMonatt* and defeat attacks (e.g., privilege escalation) caused by the vulnerabilities of the original virtualized system other than *CloudMonatt*, we can exploit some secure architectures. On the one hand, we can build *CloudMonatt* cloud servers upon security-aware systems which are designed and verified to eliminate potential vulnerabilities [52], [53], [54], [55]. On the other hand, we can use Bastion-like [56], [57] features for further protection of trusted software modules in a commodity hypervisor by creating secure enclaves for these modules. Alternatively, if the `Trust Module` is implemented in hardware, since some hardware extensions are designed for the `Trust Module`, it can be placed into a separate chip. This achieves hardware isolation between the `Trust Module` and the CPU cores.

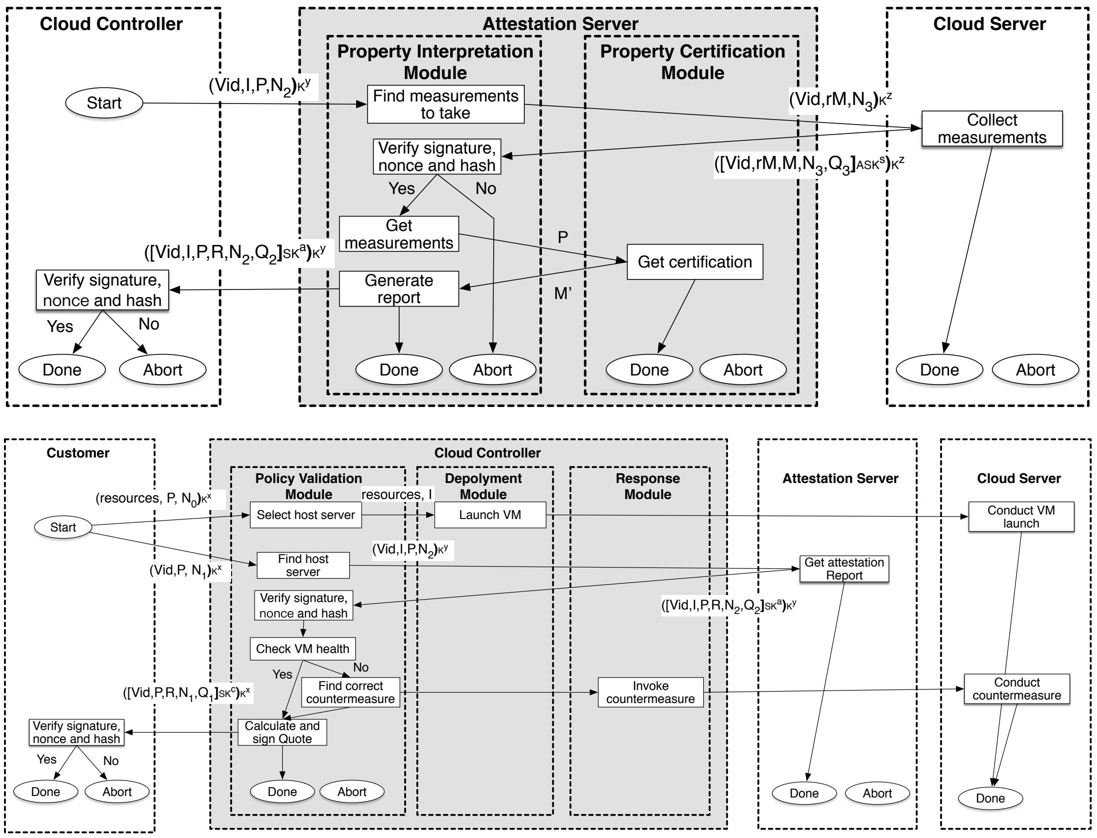


Fig. 9: Internal protocol (interactions) in the Cloud Controller

In addition, the `Monitor Module` and the `Trust Module` should be correctly designed and implemented. A correct `Monitor Module` indicates that the integrity of the measurements is guaranteed, i.e., adversaries cannot tamper with VMs' measurements collected by the `Monitor Module` before they are written into the `Trust Module`. A correct `Trust Module` indicates that the integrity of the measurements is guaranteed, i.e., adversaries cannot tamper with the measurements when they are stored in the `Trust Evidence Registers`. It also indicates that the confidentiality and integrity of the cryptographic keys are guaranteed so adversaries cannot steal or compromise the keys. So it is necessary to verify the functional correctness of these software/hardware modules. We can verify a hardware module (e.g., `Trust Module`) in two steps. We first do a functional verification, i.e., check if the hardware design has any bugs or flaws. We can use some tools to achieve this functional verification, e.g., SpyGlass [58] from Synopsys, HAL [59] from Cadence, etc. Then we do a security verification, i.e., check if the confidentiality and integrity of critical data are protected. We can use Gate-Level Information Flow Tracking (GLIFT) [60], [61], [62], [63] to formally verify these policies in the hardware design. Similarly, we can verify a software module (e.g., `Monitor Module`) in two steps. We first check if there are software bugs and vulnerabilities (e.g., buffer overflow, memory leaks) in the implementation. There are state-of-the-art tools to achieve this, e.g., Static Code Analyzer from HP Fortify Software [64], CodeSonar from GrammaTech [65], Secure Programming Lint [66], etc. Then we can use Information Flow Tracking [67], [68], [69] to verify if the security policies are enforced in the module.

5.5.2 Cloud Controller and Attestation Server Protection

In the Cloud Controller and the Attestation Server, all the critical modules should be well protected. Although it is difficult to remove all the security vulnerabilities in these central servers, there are multiple ways to enhance the security of these servers and reduce the attack surface, considering there are just a small number of such servers in the cloud system.

First, we can harden the privileged software in these servers. For instance, we can exploit formally verified operating systems [52], [54] as the host OS in the server. Besides, since these servers do not need to host guest VMs, we can remove the hypervisor layer from these servers. This can reduce the code base and potential vulnerabilities.

Second, we can provide isolations on these servers so even the server has vulnerabilities, attackers have no means to intrude into the server and exploit them. We can disable scheduling guest VMs on these servers. This eliminates the possibility that an attacker launches VMs on these servers, conducts privilege escalation attacks to host privilege and then compromises the software entities in the host OS. We can establish firewalls on these servers to prevent untrusted entities from accessing these servers. These are common security measures adopted by public cloud providers. For module-level isolation, we can use Bastion or Intel SGX to protect these software modules, as for the cloud server.

Third, we can protect the execution of cloud management services. One common method is to use Mandatory Access Control to control management and attestation services, and confine interactions among different modules [70], [71]. This can prevent adversaries from compromising the critical modules in these servers. Another method is to monitor the runtime behaviors (e.g., syscall traces) of cloud activities and establish training models for these services. When an adversary compromises a service and injects malicious behaviors, the runtime behavior will deviate from the correct one and we can detect the compromised service [72].

6 Conclusions

CloudMonatt is an architecture that enables secure monitoring and attestation of security features provided by a cloud server for the customer's VMs. We first describe the design of *CloudMonatt* and show its key advances over prior work: (1) it is flexible and provides a rich set of security properties for VM attestation; (2) it bridges the semantic gap between the security properties a customer wants to request and the measurements collected from a cloud server; (3) it enables initialization as well as runtime attestation during the lifetime of the VM; (4) it defines a novel periodic attestation capability during VM runtime; (5) it provides automated responses to bad attestation results to prevent potential, or further, security breaches; (6) it is protected by secure attestation protocols with a set of cryptographic keys that must be present or established; and (7) it is readily deployable. We leverage existing cloud mechanisms and well-honed security mechanisms where possible, identifying the minimal changes needed for a cloud system to implement our *CloudMonatt* architecture on the OpenStack cloud software.

Then, we conduct security verification of *CloudMonatt* to validate the security and correctness of this security-aware cloud architecture. To achieve scalable verification of this distributed system, we split the verification task into an external part (considering the network protocols between each server) and an internal part (considering the interactions and operations inside the server). We identify the security invariants and preconditions, model and verify each part using a cryptographic checking tool, *ProVerif*. This security verification not only raises our confidence in the design, but also helps us understand which modules/servers are critical and guides us to further enhance the security of *CloudMonatt*. Future work could be designing new security mechanisms or leveraging existing secure architectures to realize the security preconditions we identified. Also, additional security properties can be defined and translated into server measurements that can be taken, and integrated into the CloudMonatt framework.

7 Acknowledgments

This work was supported in part by NSF 1218817, NSF 1526493 SaTC and SRC 2015-TS-2632.

References

[1] D. Perez-Botero, J. Szefer, and R. B. Lee, "Characterizing hypervisor vulnerabilities in cloud computing servers," in *Proceedings of the International Workshop on Security in Cloud Computing*, 2013.

[2] T. Ristenpart, E. Tromer, H. Shacham, and S. Savage, "Hey, you, get off of my cloud: exploring information leakage in third-party compute clouds," in *Proceedings of the ACM conference on Computer and communications security*, 2009.

[3] Y. Zhang, A. Juels, M. K. Reiter, and T. Ristenpart, "Cross-vm side channels and their use to extract private keys," in *Proceedings of the ACM conference on Computer and communications security*, 2012.

[4] V. Varadarajan, T. Kooburat, B. Farley, T. Ristenpart, and M. M. Swift, "Resource-freeing attacks: Improve your cloud performance (at your neighbor's expense)," in *Proceedings of the ACM Conference on Computer and Communications Security*, 2012.

[5] T. Zhang, Y. Zhang, and R. B. Lee, "Dos attacks on your memory in the cloud," in *ACM on Asia Conference on Computer and Communications Security*, 2017.

[6] T. Zhang and R. B. Lee, "Cloudmonatt: An architecture for security health monitoring and attestation of virtual machines in cloud computing," in *ACM Intl. Symp. on Computer Architecture*, 2015.

[7] J. Szefer, *Architectures for Secure Cloud Computing Servers*. PhD thesis, Princeton University, 2013.

[8] B. Blanchet, "An efficient cryptographic protocol verifier based on prolog rules," in *Proceedings of the IEEE Workshop on Computer Security Foundations Workshop*, 2001.

[9] G. Coker, J. Guttman, P. Loscocco, A. Herzog, J. Millen, B. Hanlon, J. Ramsdell, A. Segall, J. Sheehy, and B. Sniffen, "Principles of remote attestation," *Int. J. Inf. Secur.*, vol. 10, June 2011.

[10] T. C. Group, "Tcg software stack specification." http://trustedcomputinggroup.org, Aug. 2003.

[11] R. Sailer, X. Zhang, T. Jaeger, and L. van Doorn, "Design and implementation of a tcg-based integrity measurement architecture," in *Proceedings of the Conference on USENIX Security Symposium*, 2004.

[12] T. Jaeger, R. Sailer, and U. Shankar, "Prima: Policy-reduced integrity measurement architecture," in *Proceedings of the ACM Symposium on Access Control Models and Technologies*, 2006.

[13] E. Shi, A. Perrig, and L. van Doorn, "Bind: a fine-grained attestation service for secure distributed systems," in *Proceedings of the IEEE Symposium on Security and Privacy*, 2005.

[14] A. Seshadri, M. Luk, E. Shi, A. Perrig, L. van Doorn, and P. Khosla, "Pioneer: Verifying code integrity and enforcing untampered code execution on legacy systems," in *Proceedings of the ACM Symposium on Operating Systems Principles*, 2005.

[15] T. Garfinkel, B. Pfaff, J. Chow, M. Rosenblum, and D. Boneh, "Terra: A virtual machine-based platform for trusted computing," in *Proceedings of the ACM Symposium on Operating Systems Principles*, 2003.

[16] A.-R. Sadeghi and C. Stüble, "Property-based attestation for computing platforms: Caring about properties, not mechanisms," in *Proceedings of the Workshop on New Security Paradigms*, 2004.

[17] A. Nagarajan, V. Varadharajan, M. Hitchens, and E. Gallery, "Property based attestation and trusted computing: Analysis and challenges," in *Proceedings of the International Conference on Network and System Security*, 2009.

[18] L. Chen, R. Landfermann, H. Löhr, M. Rohe, A.-R. Sadeghi, and C. Stüble, "A protocol for property-based attestation," in *Proceedings of the ACM Workshop on Scalable Trusted Computing*.

[19] J. Poritz, M. Schunter, E. Van Herreweghen, and M. Waidner, "Property attestation -scalable and privacy-friendly security assessment of peer computers," tech. rep., IBM Research, 2004.

[20] V. Haldar, D. Chandra, and M. Franz, "Semantic remote attestation: A virtual machine directed approach to trusted computing," in *Proceedings of the 3rd Conference on Virtual Machine Research And Technology Symposium - Volume 3*, VM'04, (Berkeley, CA, USA), pp. 3–3, USENIX Association, 2004.

[21] M. Alam, X. Zhang, M. Nauman, T. Ali, and J.-P. Seifert, "Model-based behavioral attestation," in *Proceedings of the 13th ACM Symposium on Access Control Models and Technologies*, SACMAT '08, 2008.

[22] E. G. Sirer, W. de Bruijn, P. Reynolds, A. Shieh, K. Walsh, D. Williams, and F. B. Schneider, "Logical attestation: An authorization architecture for trustworthy computing," in *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, SOSP '11, 2011.

[23] T. Garfinkel and M. Rosenblum, "A virtual machine introspection based architecture for intrusion detection," in *Proceedings of the Symposium on Network and Distributed Systems*, pp. 191–206, 2003.

[24] X. Jiang, X. Wang, and D. Xu, "Stealthy malware detection through vmm-based "out-of-the-box" semantic view reconstruction," in *Proceedings of the ACM Conference on Computer and Communications Security*, 2007.

[25] A. Dinaburg, P. Royal, M. Sharif, and W. Lee, "Ether: Malware analysis via hardware virtualization extensions," in *Proceedings of the ACM Conference on Computer and Communications Security*, 2008.

[26] B. Payne, M. Carbone, M. Sharif, and W. Lee, "Lares: An architecture for secure active monitoring using virtualization," in *Proceedings of the IEEE Symposium on Security and Privacy*, May 2008.

[27] B. Dolan-Gavitt, T. Leek, M. Zhivich, J. Giffin, and W. Lee, "Virtuoso: Narrowing the semantic gap in virtual machine introspection," in *Proceedings of the IEEE Symposium on Security and Privacy*, 2011.
[28] Y. Fu and Z. Lin, "Space traveling across vm: Automatically bridging the semantic gap in virtual machine introspection via online kernel data redirection," in *Proceedings of the IEEE Symposium on Security and Privacy*, 2012.
[29] S. Berger, R. Cáceres, K. A. Goldman, R. Perez, R. Sailer, and L. van Doorn, "vtpm: Virtualizing the trusted platform module," in *Proceedings of the Conference on USENIX Security Symposium*, 2006.
[30] P. England and J. Loeser, "Para-virtualized tpm sharing," in *Proceedings of the International Conference on Trusted Computing and Trust in Information Technologies: Trusted Computing - Challenges and Applications*, 2008.
[31] A.-R. Sadeghi, C. Stüble, and M. Winandy, "Property-based tpm virtualization," in *Proceedings of the 11th International Conference on Information Security*, 2008.
[32] V. Scarlata, C. Rozas, M. Wiseman, D. Grawrock, and C. Vishik, "Tpm virtualization: Building a general framework," in *Trusted Computing*, Vieweg+Teubner, 2008.
[33] M. Velten and F. Stumpf, "Secure and privacy-aware multiplexing of hardware-protected tpm integrity measurements among virtual machines," in *Proceedings of the International Conference on Information Security and Cryptology*, 2013.
[34] J. Schiffman, T. Moyer, H. Vijayakumar, T. Jaeger, and P. McDaniel, "Seeding clouds with trust anchors," in *Proceedings of the ACM Workshop on Cloud Computing Security Workshop*, 2010.
[35] N. Santos, R. Rodrigues, K. P. Gummadi, and S. Saroiu, "Policy-sealed data: A new abstraction for building trusted cloud services," in *Proceedings of the Conference on USENIX Security Symposium*, 2012.
[36] E. Brickell, J. Camenisch, and L. Chen, "Direct anonymous attestation," in *Proceedings of the 11th ACM Conference on Computer and Communications Security*, CCS '04, 2004.
[37] T. C. Group, "Tpm library specification." http://www.trustedcomputinggroup.org/tpm-library-specification/, Oct. 2014.
[38] L. Chen and J. Li, "Flexible and scalable digital signatures in tpm 2.0," in *ACM Conference on Computer and Communications Security*, 2013.
[39] F. Stumpf, O. Tafreschi, P. Röder, C. Eckert, *et al.*, "A robust integrity reporting protocol for remote attestation," in *Workshop on Advances in Trusted Computing*, 2006.
[40] L. Chen, H. Löhr, M. Manulis, and A.-R. Sadeghi, "Property-based attestation without a trusted third party," in *Proceedings of the 11th International Conference on Information Security*, ISC '08, 2008.
[41] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield, "Xen and the art of virtualization," *ACM SIGOPS Operating Systems Review*, 2003.
[42] "Intel 64 and ia-32 architectures software developer's manual, volume 3: System programming guide." http://www.intel.com/content/www/us/en/processors/architectures-software-developer-manuals.html.
[43] "Arm cortex-a9 technical reference manual, revision r2p0." http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.ddi0388e/BEHEDIHI.html.
[44] "Libvmi." http://libvmi.com.
[45] "xentrace: capture xen trace buffer data." https://linux.die.net/man/8/xentrace.
[46] D. Dolev and A. C. Yao, "On the security of public key protocols," tech. rep., Stanford University, 1981.
[47] J. Chen and G. Venkataramani, "Cc-hunter: Uncovering covert timing channels on shared processor hardware," in *Proceedings of the IEEE International Symposium on Microarchitecture*, 2014.
[48] "Openstack cloud software." http://www.openstack.org/.
[49] "Openattestation project." https://wiki.openstack.org/wiki/OpenAttestation.
[50] M. Strasser and H. Stamer, "A software-based trusted platform module emulator," in *Trusted Computing-Challenges and Applications*, Springer, 2008.
[51] B. Blanchet, "Security protocol verification: Symbolic and computational models," in *International Conference on Principles of Security and Trust*, 2012.
[52] G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, T. Sewell, H. Tuch, and S. Winwood, "sel4: Formal verification of an os kernel," in *ACM SIGOPS Symposium on Operating Systems Principles*, 2009.
[53] A. Vasudevan, S. Chaki, L. Jia, J. McCune, J. Newsome, and A. Datta, "Design, implementation and verification of an extensible and modular hypervisor framework," in *IEEE Symposium on Security and Privacy*, 2013.
[54] R. Gu, Z. Shao, H. Chen, X. Wu, J. Kim, V. Sjöberg, and D. Costanzo, "Certikos: An extensible architecture for building certified concurrent os kernels," in *USENIX Conference on Operating Systems Design and Implementation*, 2016.
[55] A. Vasudevan, S. Chaki, P. Maniatis, L. Jia, and A. Datta, "überspark: Enforcing verifiable object abstractions for automated compositional security analysis of a hypervisor," in *USENIX Security Symposium*, 2016.
[56] D. Champagne and R. Lee, "Scalable architectural support for trusted software," in *Proceedings of the International Symposium on High Performance Computer Architecture*, 2010.
[57] D. Champagne, *Scalable Security Architecture for Trusted Software*. PhD thesis, Princeton University, 2010.
[58] "Spyglass lint: Early design analysis for logic designers." https://www.synopsys.com/verification/static-and-formal-verification/spyglass/spyglass-lint.html.
[59] "Jaspergold automatic formal linting app." https://www.cadence.com/content/cadence-www/global/en_US/home/tools/system-design-and-verification/formal-and-static-verification/jasper-gold-verification-platform/jaspergold-automatic-formal-linting-app.html.
[60] X. Li, M. Tiwari, J. K. Oberg, V. Kashyap, F. T. Chong, T. Sherwood, and B. Hardekopf, "Caisson: A hardware description language for secure information flow," in *ACM Conference on Programming Language Design and Implementation*, 2011.
[61] X. Li, V. Kashyap, J. K. Oberg, M. Tiwari, V. R. Rajarathinam, R. Kastner, T. Sherwood, B. Hardekopf, and F. T. Chong, "Sapper: A language for hardware-level security policy enforcement," in *International Conference on Architectural Support for Programming Languages and Operating Systems*, 2014.
[62] A. Ferraiuolo, R. Xu, D. Zhang, A. C. Myers, and G. E. Suh, "Verification of a practical hardware security architecture through static information flow analysis," in *International Conference on Architectural Support for Programming Languages and Operating Systems*, 2017.
[63] D. Zhang, Y. Wang, G. E. Suh, and A. C. Myers, "A hardware design language for timing-sensitive information-flow security," in *International Conference on Architectural Support for Programming Languages and Operating Systems*, 2015.
[64] H. P. Enterprise, "Fortify static code analyzer." https://saas.hpe.com/en-us/software/sca.
[65] GrammaTech, "Codesonar - static analysis sast software." https://www.grammatech.com/products/codesonar.
[66] D. Larochelle and D. Evans, "Statically detecting likely buffer overflow vulnerabilities," in *USENIX Security Symposium*, 2001.
[67] A. C. Myers, "Jflow: Practical mostly-static information flow control," in *ACM Symposium on Principles of Programming Languages*, 1999.
[68] V. Simonet and I. Rocquencourt, "Flow caml in a nutshell," in *Applied Semantics II workshop*, pp. 152–165, 2003.
[69] P. Li and S. Zdancewic, "Encoding information flow in haskell," in *IEEE Workshop on Computer Security Foundations*, 2006.
[70] Y. Sun, G. Petracca, and T. Jaeger, "Inevitable failure: The flawed trust assumption in the cloud," in *ACM Workshop on Cloud Computing Security*, 2014.
[71] W. K. Sze, A. Srivastava, and R. Sekar, "Hardening openstack cloud platforms against compute node compromises," in *ACM on Asia Conference on Computer and Communications Security*, 2016.
[72] Y. Sun, G. Petracca, T. Jaeger, H. Vijayakumar, and J. Schiffman, "Cloud armor: Protecting cloud commands from compromised cloud services," in *IEEE International Conference on Cloud Computing*, 2015.

Tianwei Zhang is a software development engineer at Amazon Web Services, EC2. His research interests include cloud computing, security detection, and mitigation. He received a PhD in electrical engineering from Princeton University in August 2017. This work was done while he was a PhD student at Princeton.

Ruby B. Lee is the Forest G. Hamrick Professor in the Department of Electrical Engineering at Princeton University. Her research interests include security-aware computer architecture, secure cloud computing, smartphone security, side channels, improving security without sacrificing performance, and security validation. Lee received a PhD in electrical engineering from Stanford University. She is a Fellow of IEEE and ACM and is on the advisory board of IEEE Micro.